Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 4 (2011) 261–270

Procedia Computer Science

International Conference on Computational Science, ICCS 2011

A Multilevel Parallelism Support for Multi-Physics Coupling

Fang Liu[1,*], Masha Sosonkina[*]

Scalable Computing Lab, USDOE Ames Laboratory/Iowa State University, Ames, IA, U.S.A 50010

Abstract

A new challenge in scientific computing is to merge existing simulation models to create new higher fidelity combined (often multi-level) models. While this challenge has been a driving force in climate modeling for nearly a decade, fusion energy and space weather modeling are starting just now to integrate different sub-physics into a single model. Hence, the demand for novel software paradigms and tools increases drastically. A programming style that mixes task and data parallelism and enables concurrent execution of independent tasks on disjoint processor subsets is called multi-level parallelism. Combined models naturally map into this style, such that sub-models run simultaneously on different processor subgroups. In authors' previous work, software interfaces supporting the model coupling based on component representations are proposed and shown to successfully combine multi-physics packages via an inter-model solver. In this paper, the inter-model solver, called *Coupler*, is extended for the execution in multiple processes rather than as a single process. In essence, the multiple program multiple data paradigm is applied to multi-physics coupling. A pure C++ implementation has been developed to bypass the application adaptation to the Common Component Architecture (CCA) framework used in the previous work and to generalize the proposed approach.

Keywords: Parallel model coupling; Multi-physics modeling; Multiple Program Multiple Data programming

1. Introduction

Going towards exascale, modern computational platforms are rapidly increasing in size and heterogeneity. Hence, applications must find additional parallelism to effectively utilize the deep hierarchies of processing elements (PE) in the cutting-edge computing platforms. To enable application scientists to concentrate on the former, computer scientists and engineers may focus on mapping the application to the architecture at hand.

Many parallel applications are developed using the popular Single Program Multiple Data (SPMD) model. However, SPMD program is hard to understand and to change for large and complex applications, especially with heterogeneous computations requiring irregular or *a priori* unknown communication patterns. The Multiple Program Multiple Data (MPMD) model is preferred in this case, mainly because MPMD programs are more loosely-coupled than those written using the SPMD model. The MPMD model is more suitable for a multi-physics coupling application since it

[*] *Email address:* fangliu@scl.ameslab.gov (Fang Liu)
[1] Corresponding author

1877–0509 © 2011 Published by Elsevier Ltd. Selection and/or peer-review under responsibility of Prof. Mitsuhisa Sato and Prof. Satoshi Matsuoka
doi:10.1016/j.procs.2011.04.028

may seamlessly link existing programs. Most parallel programming systems are based either on task parallelism or on data parallelism. Task parallelism allows individual tasks to communicate and synchronize with each other through message passing or other mechanisms while data parallelism applies the same operation in parallel to different elements of a data set [1]. When a single parallel activity does not scale sufficiently, it is desirable to to launch many distinct smaller parallel tasks, each using a subset of the available processors. This style of programming is called here multilevel parallelism. It mixes task and data parallelism. The independent tasks are executing on disjoint processor subsets concurrently. The concurrent execution of independent tasks can result in shorter parallel execution times than those of the consecutive task execution on the entire set of processors [2]. The advantage of integrating task and data parallelisms is the possibly better scalability when both forms of parallelism are exploited within an application.

Applications in quantum chemistry [3], *ab initio* nuclear physics [4] and sparse linear solvers [5] demonstrate potential benefit from multilevel programming style. The Fragment Molecular Orbital (FMO) for quantum chemistry from General Atomic and Molecular Electronic Structure System (GAMESS) [6] is used to understand protein structure, enzyme catalysis, the structure of the liquids, polymer and dendrimer behavior and many other problems of importance in chemical and biology engineering. In the FMO method, a molecule is divided into N fragments, and electronic structure calculations on the fragments (monomers) and fragment pairs (dimers) are performed to obtain the total energy of the molecule. The FMO method can be expended to trimers when three-body interactions are important. The computation fragments, their pairs and triple can be assigned to the different process groups while some concurrency can be exploited.

The No-Core Shell Model (NCSM) formalism in the *ab initio* nuclear structure calculations involves the construction of very large sparse matrix (called Hamiltonian) followed by its diagonalization (e.g., eigenvalue calculation), which is a very computationally-intensive process. The parallel Many Fermion Dynamics for nucleons (MFDn) codes developed at Iowa State University (ISU) uses NCSM and is part of the U.S. Department of Energy (DOE) Universal Nuclear Energy Density Functional (UNEDF) Scientific Discovery through Advanced Computing (SciDAC) project [7]. The building blocks in the MFDn demonstrate the variable degree of parallelism which include the Hamiltonian construction for 2- or 3-body nuclear interactions, diagonalization of the Hamiltonian matrix, and computation of observables. The most time consuming part of the code is to find eigenvalues of the Schrődinger equation. In the current code design, the construction of the Hamiltonian may be done several times based on different interaction parameters, thus the construction of the Hamiltonian of the next step can overlap with the diagonalization of Hamiltonian. In this way a two-level parallelism can be achieved.

The previous research work [8] designed a common interface LInear Solver Interface (LISI) which spans multiple high-performance computing (HPC) linear system solver packages. LISI is part of the Common Component Architecture (CCA) [9] effort, and it captures the commonalities among the popular solver packages. LISI is the component interface written in Scientific Interface Definition Language (SIDL) [10] for sparse linear system solvers which facilitate an integration of a solver into scientific applications. Along with component framework, such as CCA, scientific applications can dynamically adapt to the different solvers since gaining efficiency in solving linear systems makes a considerable improvement in the entire application performance. LISI has been successfully implemented with Trilinos, PETSc, and Hypre solver packages and used in fusion energy simulation, such as the extended magnetohydrodynamics code M3D [11].

2. Overview of model coupling in LISI

A new approach for a component-based multi-physics coupling scheme is presented in previous work [12], which targets a new challenge in scientific computing: to merge existing simulation models to create new, higher fidelity combined models. In Figure 1, two physics models have different problem sizes based on discretization method used, and the models are coupled through another component, called *Coupler*. Mathematically, the coupled system may be written as

$$\begin{bmatrix} A_{11} & 0 & A_{13} \\ 0 & A_{22} & A_{23} \\ A_{13}^T & A_{23}^T & A_{33} \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}, \tag{1}$$

where A_{ii} is the matrix of the discretized linear system for sub-physics i, $i = 1, 2$; the matrix A_{i3} contains interface nodes of sub-physics i needed for the other sub-physics models, such that these interface nodes are coupled via a separate sub-matrix A_{33}. The system (1) is solved with a Schur complement method [13] which first requires an independent solve with each sub-matrix A_{ii} to eliminate the unknowns x_i, $i = 1, 2$ followed by a solution of a smaller system involving A_{33}. Each physics model component and the *Coupler* component may run on the different number of processors which may prompt the exploitation of the multilevel parallelism. Both physics models can run simultaneously, and they exchange the coupling information with the *Coupler* component. The coupler may need much fewer processors compared with the two sub-models because the corresponding linear system is typically much smaller. However, sub-grouping may be needed for sub-models as well as for coupler especially when the sub-model sizes are large and many sub-models are combined so the coupler size increases.

In this paper, the MPMD paradigm is applied for sub-grouping and the need for a tool providing multi-level parallelism is justified.

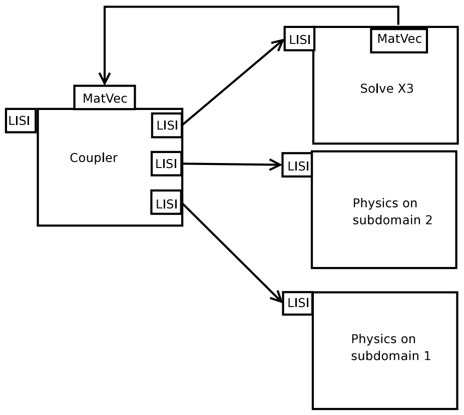

Figure 1: Component-Based Multi-Physics Coupling System

3. Extension of the *Coupler* framework

In the previous work, the *Coupler* runs only as a single process while the two physics models run on the same numbers of processors. When communicating between *Coupler* and the physics models, the augmented Message Passing Interface (MPI) [14] communicator groups are used with the assumption that *Coupler* runs in the rank-0 process. (Note that augmented groups are the groups including the process running the *Coupler* as well as those for the physics models.) Specifically, a solution is produced within the MPI communication group of a model and collected in the coupler process through the augmented MPI communicator group. Due to the assumption of the *Coupler* executing as a single process, the solution x_3 of coupled system (1), the implementation of the matrix-vector product (MatVec) is straightforward for the *Coupler* system. However, executing *Coupler* on multiple processors, the number of which may be different from the number of processors used in the physics models, is more challenging and certain implementation questions need to be answered. How should the data be passed between the MPI communication groups and between the groups with the same or different sizes? How to transfer the data efficiently?

An implementation using the Common Component Architecture CCA [9] was undertaken in the previous work. CCA is a scientific component specification and framework that allows a componentized application to plug-N-play (plug-N-play) during the runtime. However, CCA incurs a steep learning curve and an implementation overhead of adapting an existing application to CCA. In order to alleviate this burden, a multi-physics model coupling application is implemented without the CCA framework in this work.

4. Design for multilevel parallelism

To allow the coupled multi-physics applications to explore multilevel parallelism at the application level, a system is designed to support the sub-grouping in the application. The system will allow a single application to run with MPMD support such that each part of the application can run on different numbers of processes (called sub-grouping), and enable several groups to coordinate parallel activities. Currently, there are a few programming models being deployed in large scale simulations, such as MPI, Global Arrays (GA) [15, 16] and parallel virtual machine (PVM) [17].

MPI [14] is a paradigm used widely on certain classes of parallel machines, especially those with distributed memory. MPI is expected to be faster within a large multiprocessor. It has many point-to-point and collective communication options, and it has the ability to specify a logical communication topology. A number of libraries and applications are built with MPI, such as Portable, Extensible Toolkit for Scientific Computation (PETSc) [18], ScaLAPACK [19], MFDn [3] and GAMESS [6]. MPI has gained popularity in large scale simulations during the past decade.

The GA toolkit provides a shared memory style programming environment in the context of distributed array data structures (called “global arrays”). GA was designed to complement rather than substitute for the message-passing model and it allows the user to combine shared-memory and message-passing styles of programming in the same program. GA targets applications with dynamic and irregular communication patterns, often used in calculations driven by dynamic load balancing, when messaging passing becomes too complicated. NWChem [20] is one of the applications using GA as the underlying communication library.

PVM is better when applications are executed over heterogeneous networks. It has good interoperability between different hosts. PVM model is built around the *virtual machine* concept, it provides a powerful set of dynamic resource manager and process control functions which allows the development of fault tolerant applications. Since MPI is used by the two target applications (MFDn and GAMESS), the multilevel parallelism support system targets MPI for the first prototype. Later on more considerations will be given to GA and PVM.

When an MPI application starts, a default communicator, namely MPI_COMM_WORLD, is created. By default, all processes belong to the MPI_COMM_WORLD. However, MPI does not provide an easy way to create a new communicator. To introduce a new communicator into the application, MPI requires that an MPI group be created to store the neighbors in an array of rank IDs. MPI communication library mainly supports the SPMD model, so a program using collective operations cannot be re-written as a MPMD program easily. Also, to form a new communication group, an additional code has to be written for the group initialization and creation. In order to solve these two problems, two functional modules are introduced to the multilevel support system for the multi-physics coupling problem:

1. *ProcessManager* partitions the processes into the sub-groups based on the application-supplied configuration. A set of communication group identifiers are generated and queries on checking group ID are provided. Application can talk to the process manager to get partitioning information.
2. *DataConverter* is responsible for data exchange between two disjoint groups of processes, usually of different sizes. The communication pattern and synchronizations between process groups are considered.

When considering the data transferred between two sub-groups of sizes M and N, a so called $M \times N$ problem [21] may arise. The broad problem of communication between parallel entities has been traditionally termed the "$M \times N$ problem". This problem appears in a number of contexts, such as parallel data transfers, scientific data interpolation, parallel remote method invocation. In this paper, $M \times N$ problem is defined in the context of parallel data transfer. The scalability for large values of M and N implies that communications between the sub-groups should not be serialized through a single data management process and that the creation of the communication schedules should not be serialized either. Achieving scalability of the $M \times N$ problem is beyond the scope of this paper.

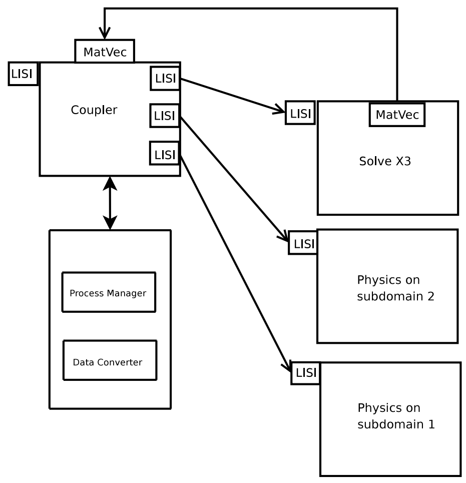

Figure 2: Multilevel Parallelism Support for Multi-Physics Coupling

Figure 2 shows how two sub-physics modules interact with the coupling component when the multi-level parallelism is supported. Two newly added modules are invoked by the *Coupler* to make the process sub-grouping and manage the data distribution. Off-loading these tasks into separate modules make the *Coupler* implementation more straightforward and easy to use. The procedure for building the communication group is simplified now and a better handling of group communication is provided.

In order to make the Application Programming Interface (API) language- and application-independent, the interface is designed with SIDL [10] as shown in Figure 3. It gives a high-level view of the interface. The package name *pmi* stands for process management infrastructure (PMI), currently four interfaces are included , besides the *ProcessManager* and the *DataConverter*, two other interfaces *Group* and *Communicator* are used to wrap the library specified communication primitives which helps to make the interface across the multiple libraries among MPI, GAi, and PVM.

```
package pmi version 1.0{
 interface ProcessManager
 {
   int dividGroups(in int ngroup, in array<int> partition, in Communicator comm);

   Group getGroupID(in int globalID);

   Group * getAllGroups();

   int isBelongToGroup(in int globalID, in Group group);

   int getLocalID(in Group group);
 }
 interface DataConverter
 {
   int getGlobalData(in array<double> ldata, out array<double> gdata, in int root, in Communicator comm);

   int sendDataFromCoupler(in array<double> data, in Communicator commA, in Communicator commB,
                   in Communicator commU, in int root);

   int receiveDataFromCoupler(in array<double> data, in Communicator commA, in Communicator commB,
                   in Communicator commU, in int root);

   int receiveDataOnCoupler(out array<double> data, in int startrow, in Communicator commA, in Communicator commB
                   in Communicator commU, in int root);

   int sendDataToCoupler(in array<double> data, in Communicator commA, in Communicator commB,
                   in Communicator commU, in int root);
 }
 interface Group
 {
    int create(in array<int> gRanklist, in int size);
    int size();
    int getAllID(out array<int> process);
 }
 interface Communicator{ int size();}
}
```

Figure 3: Interfaces of the Process Management Infrastructure defined using the Scientific Interface Definition Language (SIDL).

4.1. Coupler control flow

Figure 4 demonstrates the overall control flow for the coupled system. The major components include process manager, initialization for both coupler and sub-models, data converter, solution for coupled system and for sub-models. In the control flow, the interaction between the current coupling system and newly added models (in the red ellipse) is shown as follows:

1. Process manager is called to partition the process based on the user specified configuration.
2. During the coupler initialization, the sub-models are initialized on assigned group.
3. In the solution phase, the coupled system has to get matrix-vector perform the Schur complement matrix (see [12] for the detailed algorithm).

During each iteration, the solutions from both sub-physics models are needed. The two-sided arrows between the data converter and other models indicate the data flows. The multilevel parallelism is shown through three circled arrows which are around the solving models for the coupled system and sub-physics models. The circle 1 and circle 2 can run simultaneously, and since circle 3 is running on the different set of processes, some portion of work can be done concurrently if the algorithm is designed carefully. The data transferred between subgroups are double arrays storing the solutions of $Ax = b$ and matrix-vector product.

5. Implementation of the MPMD support

In the previous work, communication to the coupler is achieved by sending data directly to the single MPI process running coupler and by MPI_Gatherv operations on gathering data among the augmented group in which both the process running the coupler and the processes running the sub-models are included. To make the *Coupler* run in the MPMD mode, the following protocol is used

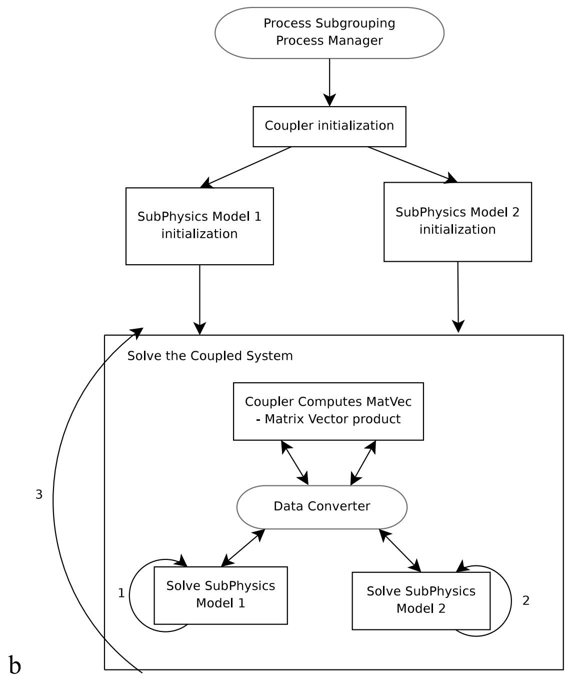

Figure 4: Control Flow for Coupler Computation

- When the coupler receives data from sub-models, sub-models first send the data to the coupler subgroup root process, then the data is distributed within the coupler subgroup.
- When the coupler sends data to sub-models, coupler first gathers the data to the coupler subgroup root process, then the data is sent to the sub-model group in which each process receives the local portion of the data.

This protocol mainly depends on the existing MPI operations. Hence, the limitation is that the data distribution calls have to provide the root process rank on which the source data resides. This approach may have potential performance bottleneck and may reach the memory limitation when the data transferred is large. The protocol can be extended to incorporate more complicated communication pattern and make the communication more efficient. Especially when the communication happens between subgroups with the different numbers of processes as in $M \times N$ problem.

5.1. Group management

The main purpose of the *ProcessManager* is to alleviate the burden on the application when building the subgroups with MPI. The *ProcessManager* may allow the program to construct the process subgroups of particular sizes, retrieve the group information, and build the inter-group process group. In this paper, the implementation is based on MPI as seen in Figure 5 presenting the API for the *ProcessManager*.

The operations include the following actions:

- Before starting a concurrent execution, the appropriate processor groups and corresponding communicators have to be established. By calling the method *dividGroups*, *ngroup* groups are generated. The integer array *partition* contains the partitions into groups, specified by the application via in a text file, *comm* can be any communicator. After calling this method, the array of MPI_Group is stored for a later query.
- Method *getAllGroups* returns the array of all the group identifiers for the running program. Currently, only one level sub-grouping is considered. Thus, all the groups are disjointed subsets of processes. In the future, when multilevel sub-grouping is added, this method may need updating such that it has an input parameter to identify the levels.

```
class ProcessManager
{
 ...
 public:
  ...
  int dividGroups(int ngroup, int * partition,
   MPI_Comm comm);

  MPI_Group getGroupID(int globalID);

  MPI_Group * getAllGroups();

  int isBelongToGroup(int globalID, MPI_Group group);

  int getLocalID(MPI_Group group);
}
```

Figure 5: MPI-based API for the *ProcessManager*.

- A process may need to check its group ID during run-time. The method *getGroupID* takes *globalID* and returns the group identifier.
- A programmer may need to determine whether or not the process belongs to a group to be able to write the corresponding implementation. The method *isBelongToGroup* provides this function.
- The method *getLocalID* is used to get the local identifier based on the group identifier. This method may be useful for calculating the data partitioning information internally.

The *ProcessManager* simplifies the procedure for building the communication group. This module interacts with *Coupler* to create the process sub-grouping. Then, the *Coupler* will start each sub-physics module on the corresponding process subgroup.

5.2. Data Converter

In the MPMD programming model, tasks have different programs to execute but usually need to exchange or share some data. MPI provides API functions for distributing data to different processes but the programmer still has to write the code for the data distribution for the designed communication pattern. The *DataConverter* is used for transferring the data between the *Coupler* and sub-models. Its main goal is to simplify the communications between subgroups. The methods are abstracted from the current coupling system. The interface functions are shown in Figure 6.

Since this data converter targets the multi-physics coupling problem, the operations are named with data flow direction associated with *Coupler*. The operations include the following actions:

- The method *getGlobalData* collects the distributed data to a specified process within a group: the local portion of data is specified and global data is only valid on the *root* process. This routine is used by the *Coupler* to prepare the data for sending to sub-models.
- The method *sendDataFromCoupler* is called by the processes running *Coupler* to send the data from the *Coupler* in *commA* to the sub-physics residing in *commB*. *commU* is the union communicator between two subgroups. In this design, the global data is first gathered in the root process in *commA* group, then is redistributed to the process in *commB* group.
- The method *receiveDataFromCoupler* is called by processes running sub-models to receive the data from *Coupler*. Each process receives the local portion of the global data.
- The method *receiveDataOnCoupler* allows every process in the coupler subgroup to receive a local copy of the data from sub-models.
- The method *sendDataToCoupler* is called by sub-models to send the their portions of the global data back to the coupler.

These are core methods for multi-physics coupling system, the API of which can always be extended to meet the new requirements, such as those needed by applications from Section 1.

```
class DataConverter
{
 ...
 public:
  ...
  int getGlobalData(double * ldata, int llen,
   double ** gdata,int root, MPI_Comm comm);

  int sendDataFromCoupler(double * data, int length,
   MPI_Comm commA, MPI_Comm commB, MPI_Comm commU,
   int root);

  int receiveDataFromCoupler(double * data, int length,
   MPI_Comm commA, MPI_Comm commB, MPI_Comm commU,
   int root);

  int receiveDataOnCoupler(double ** data, int startrow,
   int length, MPI_Comm commA, MPI_Comm commB,
   MPI_Comm commU, int root);

  int sendDataToCoupler(double * data, int length,
   MPI_Comm commA, MPI_Comm commB, MPI_Comm commU,
   int root);
}
```

Figure 6: API for the *DataConverter*.

6. Testing

The test case used is the same as the one from [12]. Two physics models are based on the following two-dimensional Partial Differential Equations (PDEs),

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f \tag{2}$$

with Dirichlet boundary conditions, discretized with five-point centered finite-difference scheme on $n_x \times n_y$ grid. Where $f = (2.0 - 6.0 * x - x^2) * sin(y)$ and boundary condition $b = x * x * sin(y)$.

Figure 7 shows the test problem. Domain 1 is on the left side with domain boundary $[0, 1]\times[0, 1]$ while domain 2 (right) has the boundary $[1, 2] \times [0, 1]$. The interface nodes align vertically at $x = 1$ between two domains. The interface boundary domain is discretized with one dimensional PDEs. In the test, the number of mesh points are chosen for a single direction. For example, when n nodes is used for one direction, the total mesh points will be $n \times n$.

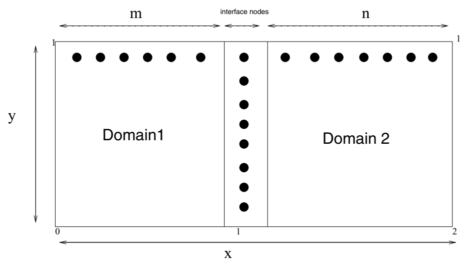

Figure 7: Test problem setup

Sub-physics 1 runs with Trilinos AZTECOO solver with maximum iteration number 500 and tolerance of $1.0e^{-6}$. The solver method BICGSTAB and preconditioner method Jacobi are used. Sub-physics 2 runs High Performance Preconditioners (HYPRE) BOOMERAMG solver with the maximum of 30 iterations and tolerance of $1.0e^{-6}$. All the other parameters are set to default. The coupled system is solved with PETSc BICGSTAB method with the maximum of 500 iterations and tolerance of $1.0e^{-6}$.

The tests have been performed at Ames Laboratory on the dual-core 2 Ghz Intel Xeon processor cluster having four nodes with two processor each. In Table 1, *size* is the mesh size × mesh size for each sub-physics domain; *its* is the iteration number for the solution of the coupled system; *residual* is the final residual. Table 1 shows the tests with three different sizes for the tests: 50, 100 discretized nodes in each dimension. The tests run on 6 processes total, sub-physics 1, sub-physics 2 and *Coupler* all run on 2 processes. The test is also done for the problem on mesh sizes of 200. The tests run on 10 processes total, sub-physics 1 and sub-physics 2 run on 4 processes and *Coupler* runs

size	*its*	*residual*
50×50	129	$3.72e^{-6}$
100×100	145	$7.98e^{-8}$

Table 1: Experimental Results on Two Sub-physics Coupling: Coupler, Sub-physics 1 and Sub-physics 2 all run on 2 processes.

on 2 processes. The test performs 789 iterations with $4.24e^{-6}$ residual. These two sets of tests are to demonstrate that *Coupler* can run on multiple processes and that *Coupler* and sub-physics models can run on different numbers of processes. The results show that the tests are successful.

7. Related Work

Much effort has been put into exploring the combination of task and data parallelism. Programming language support [1] was investigated on both forms of parallelism within a single application, addressing the support of the SPMD and MPMD style programs through language features. A library support for hierarchical multi-processor tasks is presented in [22]. It considers modular programming with hierarchically structured multi-processor tasks on top of SPMD tasks for distributed memory machines. This work decomposes the set of processors into a hierarchical group structure onto which the tasks are mapped. The library is built on top of MPI, has an easy-to-use interface, and leads to only a small overhead while allowing static planning and dynamic restructuring. Tools, such as Model Coupling Toolkit (MCT) [23], provide a common utility to couple the climate models with the focus on the grid data exchange between the models. MCT addresses the inter-model, parallel communication requirements of Community Climate System Model (CCSM) and intra-model communication requirements of a distributed-memory parallel coupler. MCT is more domain-specific or domain-inspired solutions that the coupling system proposed here and is entirely written in Fortran90. A programming support for MPMD was built for graph-oriented programming in [24], where an abstract layer on top of MPI provides the support to programmer for building MPMD applications through a library of functions for communications, distributed shared data, and mapping of applications to underlying processors for execution.

8. Conclusion

In this paper, the MPMD support for coupling of multi-physics codes is investigated. The work extends the authors' previous multi-physics coupling work by enabling the *Coupler* component to run on the multiple processes instead of a single process. This work nears the coupling model to the real world application in which all participated components are running on multiple processes. The support system for multilevel parallelism is proposed with two functional modules, *ProcessManager* and *DataConverter*. The process manager deals with the subgroup partitioning while the data converter transfers the data between the different sets of process groups. By adding these two models, the coupling system not only has a simpler code structure, which allows programmer to focus on the program logic, but also the system becomes more realistic. For the simplicity of the prototype implementation, the current version utilizes a simple communication protocol in which the data is always sent to root process within the group before it is distributed to the other groups. The protocol potentially has some performance bottlenecks. In the future, another module may be added to take care of the communication between two groups in the case of the $M \times N$ problem of data sharing. Also, the proposed implementation is based on MPI. The future work may consider other parallel programming paradigms, such as GA and PVM.

References

[1] H. E. Bal, M. Haines, Approaches for integrating task and data parallelism, IEEE Concurrency 6 (1998) 74–84. doi:10.1109/4434.708258. URL `http://portal.acm.org/citation.cfm?id=614062.614113`

[2] M. Khnemann, T. Rauber, G. Rnger, Performance modelling for task-parallel programs, in: In Proc. of the Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS 2002, 2002, pp. 148–154.

[3] J. P. Vary, The Many-Fermion Dynamics Shell-Model Code, Iowa State University (1992).

[4] M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Gordon, J. H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. Su, T. L. Windus, M. Dupuis, J. A. Montgomery, Jr., General atomic and molecular electronic structure system, J. Comput. Chem. 14 (11) (1993) 1347–1363. doi:http://dx.doi.org/10.1002/jcc.540141112.

[5] Hypre: Scalable Linear Solvers : High Performance Preconditioners, `http://www.llnl.gov/CASC/linear\_solvers/` (April 2008).
URL `http://www.llnl.gov/CASC/linear\_solvers/`

[6] M. S. Gordon, M. W. Schmidt, Advances in electronic structure theory: Gamess a decade later, Theory and Applications of Computational Chemistry:the first forty years, C.E.Dykstra, G.Frenking, K.S.Kim, G.E.Scuseria (Editors).

[7] P. Sternberg, E. G. Ng, C. Yang, P. Maris, J. P. Vary, M. Sosonkina, H. V. Le, Accelerating configuration interaction calculations for nuclear structure, in: SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, IEEE Press, Piscataway, NJ, USA, 2008, pp. 1–12.

[8] F. Liu, R. Bramley, CCA-LISI: On Designing a CCA Parallel Sparse Linear Solver Interface, in: Proc. 21th Int'l Parallel & Distributed Processing Symp.(IPDPS), ACM/IEEE Computer Society, 2007, 10 pages.

[9] CCA-Forum, The DOE Common Component Architecture Project, `http://www.cca-forum.org/` (April 2008).

[10] T. Dahlgren, T. Epperly, G. Kumfert, J. Leek, Babel User's Guide, CASC, Lawrence Livermore National Laboratory, Livermore, CA, babel-0.11.0 Edition (2005).
URL `http://www.llnl.gov/CASC/components/docs/users_guide.pdf`

[11] W. Park, E. V. Belova, G. Y. Fu, X. Z. Tang, H. R. Strauss, L. E. Sugiyama, Plasma Simulation Studies Using Multilevel Hhysics Models, Vol. 6, AIP, 1999, pp. 1796–1803. doi:10.1063/1.873437.
URL `http://link.aip.org/link/?PHP/6/1796/1`

[12] F. Liu, M. Sosonkina, R. Bramley, A new approach: Component-based multi-physics coupling through cca-lisi, Lecture Notes in Computer Science 6017/2010 (2010) 503–518.

[13] F. Zhang, The Schur Complement and Its Application, Numerical Methods and Algorithms, Springer, 2005.

[14] M. P. I. Forum, MPI: A Message-Passing Interface Standard, International Journal of Supercomputer Applications and High Performance Computing 8 (3/4) (Fall-Winter 1994) 159–416.

[15] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, E. Aprà, Advances, applications and performance of the global arrays shared memory programming toolkit, Int. J. High Perform. Comput. Appl. 20 (2006) 203–231. doi:10.1177/1094342006064503.
URL `http://portal.acm.org/citation.cfm?id=1125980.1125985`

[16] Global Arrays Webpage, `http://www.emsl.pnl.gov/docs/global` (Jan 2011).

[17] Parallel Virtual Machine Webpage, `http://www.csm.ornl.gov/pvm` (Jan 2011).

[18] PETSc: Portable Extensibel Toolkit for Scientific Computation, `http://www-unix.mcs.anl.gov/petsc/petsc-as/` (April 2008).
URL `http://www-unix.mcs.anl.gov/petsc/petsc-as/`

[19] ScaLAPACK Home Page, `http://www.netlib.org/scalapack/scalapack_home.html` (May 2009).
URL `http://www.netlib.org/scalapack/scalapack_home.html`

[20] M. Valiev, E. Bylaska, N. Govind, K. Kowalski, T. Straatsma, H. V. Dam, D. Wang, J. Nieplocha, E. Apra, T. Windus, W. de Jong, Nwchem: A comprehensive and scalable open-source solution for large scale molecular simulations, Computer Physics Communications 181 (9) (2010) 1477 – 1489. doi:DOI: 10.1016/j.cpc.2010.04.018.
URL `http://www.sciencedirect.com/science/article/B6TJ5-502V6YP-2/2/2d40ddac658b388d1681698f0642`

[21] F. Bertrand, Data redistribution and remote method invocation in parallel component architectures, Ph.D. thesis, Indiana University (2005).

[22] T. Rauber, G. Runger, Library support for hierarchical multi-processor tasks, SC Conference 0 (2002) 45. doi:http://doi.ieeecomputersociety.org/10.1109/SC.2002.10064.

[23] J. Larson, R. Jacob, E. Ong, The Model Coupling Toolkit: A New Fortran90 Toolkit for Building Multiphysics Parallel Coupled Models, International Journal of High Performance Computing Application 19 (2005) 277–292.

[24] F. Chan, J. Cao, A. T. Chan, M. Guo, Programming Support for MPMD Parallel Computing in ClusterGOP, IEICE Transactions on Information and Systems E-87D (2004) 1693–1702.